CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

NL 339 (r. 85/09)

National Library of Canada
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada

TC –

ISBN 0-315-21218-7

CANADIAN THESES ON MICROFICHE SERVICE – SERVICE DES THÈSES CANADIENNES SUR MICROFICHE

PERMISION TO MICROFILM – AUTORISATION DE MICROFILMER

• Please print or type – Écrire en lettres moulées ou dactylographier

AUTHOR – AUTEUR

Full Name of Author – Nom complet de l'auteur

BRIAN CHARLES WILKERSON

Date of Birth – Date de naissance

JUNE 9, 1959

Canadian Citizen – Citoyen canadien

☐ Yes / Oui ☒ No / Non

Country of Birth – Lieu de naissance

UNITED STATES OF AMERICA

Permanent Address – Résidence fixe

3906 NE. SKIDMORE
PORTLAND, OR 97211
U.S.A.

THESIS – THÈSE

Title of Thesis – Titre de la thèse

SMALLTALK-80: ANOTHER VIEW

Degree for which thesis was presented
Grade pour lequel cette thèse fut présentée

MASTERS OF SCIENCE

Year this degree conferred
Année d'obtention de ce grade

1985

University – Université

UNIVERSITY OF ALBERTA

Name of Supervisor – Nom du directeur de thèse

DUANE SZAFRON

AUTHORIZATION – AUTORISATION

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to **microfilm** this thesis and to **lend** or **sell copies of the film.**

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de **microfilmer** cette thèse et de **prêter** ou de **vendre des exemplaires du film.**

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

ATTACH FORM TO THESIS – VEUILLEZ JOINDRE CE FORMULAIRE À LA THÈSE

Signature

B. C. Wilkerson

Date

APRIL 24, 1985

Canada

The University of Alberta

Smalltalk-80: Another View.

by

(C) Brian Wilkerson.

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Masters of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1985

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Brian C. Wilkerson

TITLE OF THESIS: Smalltalk-80: Another View

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1985

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Brian C. Wilkerson

Permanent Address:

3906 NE Skidmore
Portland, Oregon
U.S.A. 97211

Dated April 13, 1985

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Smalltalk-80: Another View** submitted by **Brian C. Wilkerson** in partial fulfillment of the requirements for the degree of **Master of Science**.

Duane Szafron

Supervisor

Clifford Addison

Jonathan Schaeffer

Date April 22/85

Abstract.

Smalltalk is one of the most interesting and exciting developments in the area of programming languages. It has had repercussions throughout the field of computing science. The influence, ramifications and possible uses of Smalltalk make it a required topic of study for all computing scientists. The purpose of this paper is to provide a critical examination of the important features of Smalltalk. We will also examine some of the possible future directions for ongoing research in this field. While this paper is not meant to be a general introduction to Smalltalk it does not require of the reader any prior exposure to Smalltalk.

Acknowledgements.

I would like to thank my supervisor, Dr. Duane Szafron, both for the technical and the moral support he has provided, and especially for the long hours of discussion that gave this paper some real substance, and provided many of the ideas in it. I would also like to thank Dave Clemens and the other members of the ECS group at Tektronix for a very rewarding and informative summer. And I would like to give a special thanks to Karen Cunningham, who first introduced me to the beauty of Smalltalk, and whose inspiration, so many years ago, prompted me to study computing science.

Table of Contents

Chapter 1. Introduction. 1

Chapter 2. What is Smalltalk-80? 4

Chapter 3. The History of Smalltalk. 8

3.1 The Dream. 8

3.2 Xerox's Research. 9

3.3 The Simula Influence. 9

3.4 The Actor Model. 10

3.5 Actors Influence Smalltalk. 11

Chapter 4. Smalltalk, The Environment. 13

4.1 The Nature of the Interface. 13

4.1.1 Menus. 15

4.1.2 Windows. 15

4.2 The Philosophy of the Interface. 16

Chapter 5. The Smalltalk Language. 18

5.1 Objects. 18

5.2 Message Sending. 19

5.3 Methods. 22

5.4 The Concept of Class. 23

5.5 Inheritance. 25

5.5.1 Collection Classes: An Example of Inheritance. 27

5.5.2 Problems with Inheritance. 31

Chapter 6. The Smalltalk Metaphor. 32

6.1 Objects As Actors. 32

6.2 Object-oriented Programming. 33

Chapter 7. Implementation. 36

7.1 A Virtual System. 37

7.1.1 The Virtual Machine. 38

7.1.2 The Virtual Image. 39

7.2 Object Memory. 39

7.2.1 Garbage Collection. 41

7.3 Bytecodes. 42

7.4 Primitive Methods. 42

7.5 Influence Of Implementation On System. 43

Chapter 8. The Smalltalk Reality. 47

8.1 Where It Falls Short. 47

8.1.1 Problems With the Language. 47

8.1.1.1 Cascaded Messages. 47

8.1.1.2 Assignment. 48

8.1.1.3 Variables. 49

8.1.1.4 A Solution. 51

8.1.2 Problems With the Class Descriptions. 51

8.1.2.1 The Class *BitBlt*. 52

8.1.2.2 Scalars and Subranges. 53

8.1.2.3 Private Messages. 56

8.1.3 Problems With the Implementation. 56

8.2 Where They Did It Right. 57

8.2.1 Benefits of the Class Descriptions. 57

8.2.2 Benefits of the Metaphor. 58

Chapter 9. Future Directions. 60

9.1 Strongly Typed Smalltalk. 60

9.1.1 Problems with Strong Typing. 61

9.2 From Programming Environment To Operating System. 61

9.3 Parallel Computation and Multiple Processors. 63

9.4 Introducing Color. 64

9.5 Introducing Sound. 64

Chapter 10. Conclusions. 65

References. 66

Appendix A. A Partial Hierarchy of Smalltalk Classes. 69

Appendix B. The Definition of the Class Bag. 72

List of Figures.

Figure 2.1. Three aspects of Smalltalk. 5

Figure 5.1. The Class Structure of Smalltalk. 25

Figure 5.2. Full Metaclass Hierarchy. 27

Figure 5.3. The Smalltalk Hierarchy of Collection Classes. 30

Figure 7.1. The Virtual System. 38

Chapter 1.

Introduction.

Smalltalk is one of the most interesting and exciting developments in the area of programming languages. It has changed the way we look at programming by offering a new methodology for the design and analysis of programs. It also provides a richer environment for the development and use of computer programs, both by professional programmers and by casual computer users. Although these two reasons are enough to make Smalltalk a worthwhile topic of study, it is more than just a new programming language.

The results of research into Smalltalk and its uses have had repercussions throughout the field of computing science. Smalltalk has one of the first modern graphical user interfaces, one that strongly influenced the development of the interface on the popular Macintosh[1] microcomputer. Smalltalk provides an environment in which different styles of user interfaces can be easily studied and evaluated. Smalltalk provides a new metaphor for the expression of intelligent behavior. And finally, Smalltalk has contributed to a new object-oriented design methodology in software engineering. The influence, ramifications and possible uses of Smalltalk make it a required topic of study for all computing scientists.

The goal of this paper is to provide a critical examination of Smalltalk. We wish to examine the most important features of Smalltalk and to discuss

[1] Macintosh is a trademark of Apple Computers, Inc.

their strengths and weaknesses. By doing so we will gain a better understanding of why Smalltalk is having, and will continue to have, such a large impact on computing science.

Although this paper is a critique of Smalltalk, previous exposure to Smalltalk is not required. We will provide a brief description of those features of Smalltalk that are important to the topic at hand. This discussion is not meant to be a thorough description of Smalltalk, but it should be complete enough that the problems and benefits noted in this paper can be easily understood.[2]

We begin our brief description of Smalltalk with a discussion of the terminology used in this paper and a look at the history of Smalltalk. We then examine the various components of the Smalltalk system and examine the implementation of Smalltalk-80 in some detail. Since Smalltalk is interpreted, the implementation directly affects the run-time environment, which in turn affects the nature of the system. Therefore, a full understanding of Smalltalk requires at least a partial understanding of the implementation. We also look at alternative implementation schemes, and the effect these schemes would have on the system.

By the end of the description the reader should have a good understanding of Smalltalk and should be be familiar with most of the features of Smalltalk. The remaining sections concentrate on the factors that make each feature either good or bad. We limit the scope of this paper to an examination of those features of Smalltalk that we consider to be most important.

[2] Readers who are interested in a more thorough description of Smalltalk are directed to [Gold82].

The paper concludes with an examination of several possible future enhancements to Smalltalk. These enhancements are evaluated in turn. The evaluation is based on how well a feature meets the goals of Smalltalk, its practical or theoretic value, and the availability of the means of implementing the feature.

Chapter 2.

What is Smalltalk-80?

One of the biggest problems we face when describing a new idea or development is the lack of a widely known and accepted terminology. We can try to use existing words that are close in meaning, but this tends to hide the important distinctions. If we simply invent new words no one will know what we mean until we give a definition. Yet, if we do not fully understand the implications of a new idea, we will not be able to give a complete definition.

The terms that have become associated with Smalltalk are a perfect example of this problem. Although they are well known and frequently used, the definitions of these terms are still vague. Even the word "Smalltalk" itself is only loosely defined. Before we can evaluate Smalltalk, we must have a consistent definition of the terms used.

So, what exactly do we mean by the word "Smalltalk"? Most people who "know" Smalltalk would probably tell you that it is a programming language. If they wanted to be more precise, they would describe it as an interactive, graphical programming language. Unfortunately, this second definition can be misleading. It may imply to some that Smalltalk is used to program graphics, or that the Smalltalk programming language is represented graphically rather than by text. The latter is not true at all, and while it is easy to write graphically oriented applications in Smalltalk, that is certainly not the

only (nor the most important) use of the language.

The problem is that there is no single definition for the word "Smalltalk". As a result of the unique nature of Smalltalk, the name has come to be used for three separate but related concepts. The relationships between these concepts is illustrated in figure 2.1. We will look at each of these concepts in more detail in future chapters, but it will be helpful to have a general understanding of the word before proceeding.

One use of the word "Smalltalk" is to refer to the programming environment, or user interface, that is used. Smalltalk has a highly interactive, graphically oriented interface. Text and pictures are displayed in windows on a bit-mapped screen while a pointing device (usually a mouse) and the keyboard are used to direct the actions of the system. This particular style of user interface (the way in which these elements are used, manipulated, etc.) is sometimes referred to as "Smalltalk".

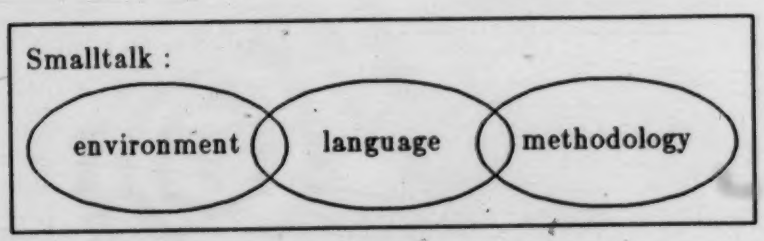

Figure 2.1 -- Three aspects of Smalltalk.

The word "Smalltalk" is also used to refer to the programming language itself. This is probably the most common use of the term. Smalltalk is an interpreted language, with dynamic binding of both variables and methods (methods correspond to procedures in procedural languages). Smalltalk also uses a totally dynamic allocation scheme, with explicit allocation, and automatic deallocation. It is the syntax and semantics of this highly dynamic language which have come to be referred to as "Smalltalk".

The third use of the term is to describe the programming methodology embodied by the programming language Smalltalk. Using the word in this way is becoming less common as the term "object-oriented" begins to take its place [Rent82]. A programmer using this methodology would begin by representing the problem in terms of an active agent. The behavior of this agent would next be defined, followed by the knowledge needed by the agent and finally the actions taken by the agent to affect the appropriate behavior. The agent is chosen in such a way that the solution to the problem can be represented as an instruction to the agent to take some action. This action usually involves asking other agents involved in the solution to take some other actions, etc. This method of problem solving is, however, sometimes still referred to as "Smalltalk".

This confusion of terms is highly undesirable. Therefore, throughout the remainder of this paper we will use the following definitions. The word "Smalltalk", unless qualified, will refer either to the programming language or to the system as a whole (the language, the methodology and the environment). The distinction between the programming language and the system

should be clear from context. If we wish to discuss the Smalltalk environment, we will simply call it "the environment". When we discuss the programming methodology, we will use the term "object-oriented methodology".

Chapter 3.

The History of Smalltalk.

Before we go on to our in-depth look at each of the three major portions of Smalltalk, we should take a quick look at Smalltalk's history. It is important to know where Smalltalk comes from if we are to understand why it is the way it is, and where it is likely to go in the future.

3.1. The Dream.

While it is impossible to point to one moment in history and say "this is when the idea of Smalltalk was born", we generally place its beginnings between 1967 and 1969. Alan Kay, who was then a Doctoral student at the University of Utah, began working on a project known as the FLEX machine. The FLEX machine was an attempt to achieve, both in hardware and in software, the prototype of a machine that Alan Kay referred to as the Dynabook [Kay77].

The Dynabook was to be a truly personal computer. It was envisioned as providing a vehicle for personal expression in a box the size of a standard notebook. The surface of the machine would contain a touch sensitive, color graphics display. The user would point to items on the screen by pressing them, and the keyboard would simply be part of the screen. It would also contain sound generators, with the possibility of "playing" the machine via a touch sensitive, piano-like keyboard. The Dynabook was to be a machine

usable by anyone, anywhere, any time.

3.2. Xerox's Research.

The FLEX machine was a step toward the goal of the Dynabook, but it was not enough to satisfy Alan Kay. In an effort to insure that the research could continue, he presented his ideas and work to a group of people at the Xerox Corporation. They were interested. In 1970, Alan joined the newly formed Learning Research Group at Xerox. The goal of the group was to continue Kay's work and to produce a Dynabook. Since the level of technology was insufficient at that point to complete the project, the group began work on what they termed an "intermediate" Dynabook.

Since the Dynabook was to be an interactive, user friendly system, which was usable by everyone, it would have to have a simple interface. Since it was to be a truly personal computer, it would have to be easy to modify the system in a consistent way without requiring system wide changes. This required that a simple but powerful programming language be created, one designed specifically to provide these features. Smalltalk was to be the language for the Dynabook.

3.3. The Simula Influence.

Of all the languages that influenced Smalltalk, Simula-68 was the first to have a major effect. In fact, Simula-68 is the source of much of the terminology associated with Smalltalk. It was derived from Simula, a simulation implementation language. The major innovation in Simula-68 was the addition of classes. The designers of Simula-68 realized that when implementing

a simulation it would be helpful to be able to define each of the "parts" of the simulation separately, with the ability to create as many copies of these parts as required. This lead to the concept of a class.

In Simula-68, a class was intended to represent either an object used in the simulation, or a process that occurred in the system being simulated. Thus, a class consisted of several related procedures, together with the data structures used and manipulated by those procedures.

The designers of Smalltalk saw two major problems with the Simula-68 definition of classes. First, Simula-68 does not enforce information hiding. This means that changes in the definition of a class cannot be localized to the implementation of the class. The second major problem is that not all classes are created equal. Classes defined by the user are more limited than system classes, and system classes can not be modified by the user.

The first implementation of Smalltalk, known as Smalltalk-72, attempted to correct those problems [Shoc79]. It also tried to use the class concept more consistently throughout the system. This lead to a system in which the data structures were given more importance than the procedures which manipulated them. The result of these efforts, while not perfect, was encouraging enough to keep the project going.

3.4. The Actor Model.

In 1973, Carl Hewitt, who was working at the MIT Artificial Intelligence Laboratory, published, with others, a paper on the subject of actors [Hewi73]. Actors are a generalization of the concept of objects in Smalltalk-72. An

actor is an independent, intelligent agent who is capable of taking actions on cue according to a predefined script. Future work on actors served to strengthen the initial concepts [Hewi77].

The analogy behind actors is an appealing one. Each actor is an individual with a right to its own privacy. Actors are allowed to communicate using the generalized concept of message passing. They may make requests of other actors, but no actor can force another actor to take actions against its will since that would be an invasion of privacy. Similarly, no actor can forcibly extract information from other actors.

This approach is precisely what is meant by data abstraction. Actors are defined only by the operations that can be performed on them (that is, the messages that can be sent to them). It implies information hiding by not allowing any actor to manipulate the state of another actor, another important feature of data abstraction. Note that this is not the same thing as data encapsulation.[3] While data encapsulation existed in Simula-68, this treatment of classes provided Smalltalk with a powerful new way of looking at data abstraction.

3.5. Actors Influence Smalltalk.

The work of Hewitt influenced the rewriting of Smalltalk. Objects were given more control over their own state, and there was an increase in the importance placed on information hiding. These changes, together with significant improvements in the syntax combined to form Smalltalk-76

[3] for a discussion of these terms see [Fair85] p. 139-140.

[Inga78], the second generation.[4]

Still not happy with the form of Smalltalk, the Learning Research Group began on the third generation which eventually became Smalltalk-80. There were minor syntactic changes in the language, but the major change came in the definition of the classes in the system, the implementation of the system, and the environment. The environment was redesigned to be easier to use, the implementation was made more efficient, and the class definitions were improved. With these changes, Xerox released Smalltalk-80 [Gold83] [Inga81a], the version we are examining in this paper.

[4] A discussion of the history of Smalltalk to this point, and a description of a dialect of Smalltalk-76 can be found in [Kade78].

Chapter 4.

Smalltalk, The Environment.

We begin our study of Smalltalk with a look at the environment [Gold82], [Gold84]. The reason for this is that the environment is the most obvious and visible part of Smalltalk. It is both graphical and interactive [Inga81b]. This graphical interaction is usually embodied in the form of a monochrome bit mapped display and a three button mouse. Since touch sensitive screens are not generally available, the system still needs to have a keyboard attached to it.

The system makes extensive use of graphical entities, such as windows, menus, etc., so that it is truly interactive. This interaction places a restrictive upper bound on the response time that can be tolerated.

4.1. The Nature of the Interface.

There are three types of graphical entities used: cursors, windows, and menus. These three types combine to form an interface that is both simple and elegant. The underlying theme is that everything the user wants should be available. By available we mean not only that it should be visible on the screen, but also that the user should not be required to "get out of" whatever they are currently using to be able to use something new. This style of user interface is known as *modeless*.

In keeping with the object-oriented approach, all graphical entities are themselves objects. The user interacts with the system by sending messages to the objects within the system. These messages are sent by selecting items from a menu. [5]

Even elements of the hardware are considered to be objects. Moving the mouse is a message to the object *Cursor* to change its position on the screen. Pressing a key on the keyboard causes a new object of type *KeyboardEvent* to be created, whose existence and state can be queried by other objects in the system.

The meaning associated with each of the three mouse buttons depends on the current position of the cursor on the screen. To help the user determine the effects of pushing a mouse button, the cursor can be asked to display a different representation of itself at different times. To promote a simple interface, the meanings of the three mouse buttons are as consistent as possible. The left-most button is used to select items. These items could include such things as pieces of text, subportions of pictures, or graphical entities like windows. The middle button is used to manipulate the contents of the window containing the cursor, or, if the cursor is not in a window, the system as a whole. This is done by bringing up a window-dependent menu from which commands are selected. The right-most button is used to manipulate the window itself. If the cursor is not in any window, this button is not used.

---

[5] While not usually necessary, it is possible to send messages not provided for in the menus by entering the Smalltalk code for the message send and executing it.

4.1.1. Menus.

The menus used in the interface are known as pop-up menus. As the name implies, these menus "pop up" at the current cursor position when a menu-producing mouse button is pressed. This should be contrasted with the alternate style of menus, called pull-down menus. In this style, the menu is represented by its name, and the user must move the cursor to the location at which the menu's name appears, and then depress the mouse button.

The pop-up menus used in the Smalltalk environment have one major advantage over pull-down menus: the user does not need to move the mouse as often (nor as far) to accomplish menu driven operations. This tends to make the interface easier to use once the location of the menus has been learned (ie. which mouse button brings up which menus), but is more difficult to learn.

Another feature of Smalltalk's menus, is that the menu always comes up with the last selected command as the default selection. This simplifies repetitive execution of commands, but may be disconcerting for those accustomed to other systems.

4.1.2. Windows.

Windows are a common way to display both text and graphics. Each window consists of two parts: a *title tab* and a *view*. The title tab contains the name of the window, and appears above the upper left hand corner of the view. The view is a rectangular region of the screen in which the information associated with the window will be displayed. In addition, there are two

kinds of views. Primitive views display data in a readable format, and compound views are those that contain sub-views. Sub-views can likewise be either primitive or compound.

Since the contents of a primitive view may be only a subportion of the total data available, each primitive view can have a scroll bar associated with it. A scroll bar is a thin region that displays the relative size and position of the data within the view, with respect to the total amount of data available. Scroll bars are only visible while the cursor is within the associated view. In addition to providing the information above, scroll bars can be used to move through the data being presented.

4.2. The Philosophy of the Interface.

It is useful to briefly examine the philosophy behind the implementation of the user interface. By doing so, the reader should get not only a better understanding of the way in which interactions with the system are viewed, but also a better feeling for the object-oriented approach to design.

Those objects that are meant to be displayed on the screen and directly manipulated by the user are composed of three parts. The first part is called the "model". The model is the data actually being displayed and changed. The second part is called the "view". The view is the object responsible for presenting the model in a format that is easy for the user to understand. The last part of a displayed object is called the "controller". The controller is responsible for translating user actions and requests into messages that are then sent to the model.

An example should make this clear. Consider a text editor (a common component of the environment). The text being edited (the characters that make up the text, the emphasis of the characters, etc.) is the model. This is the information that the user wishes to manipulate. The view controls the way in which that text is formated in the window. It is responsible for printing out the current state of the text, managing the scrolling of information within the window, etc. The controller is the object that responds to user input. It defines the meaning of pressing keys, associates menus with mouse buttons, and sends messages to the text to effect the changes requested by the user.

It is the coordination of these three parts that produces the interface seen by the user. The controller checks for user input. When the user moves the mouse to select some text, the controller first discovers from the view which text was displayed in the indicated region, then tells the text to select a portion of itself, and then tells the view that the text has been changed. This will cause the view to update the screen, providing the user with a visual feedback (namely, the highlighting of the selected text).

Chapter 5.

The Smalltalk Language.

The Smalltalk system was designed with two principal goals in mind. First, the system had to be interactive so that it would be easy to use. In order for the language to fit into the interactive environment desired, the language itself would have to be interactive. This means that changes to the Smalltalk code must be incorporated into the system immediately without the usual overhead of a lengthy compilation and linking procedure. For this reason, the Smalltalk language is usually interpreted.

The second goal was that the system must be easy to change and those changes must be automatically reflected in a consistent manner throughout the whole system. The requirement that changes be easy to make in turn requires that information be highly localized. In this way, a piece of information need only be changed in one place. For this purpose, the language designers utilized the concepts of actors and classes.

5.1. Objects.

The heart of the Smalltalk programming language is the "object".[6] An object is an abstract quantity, representing an intelligent entity that responds to certain requests. This is similar to the definition of an actor as given by Hewitt. The only difference is that objects in Smalltalk are not as

[6] Examples of how to program using objects can be found in [Alth81] and [Deut81].

restricted as Hewitt's actors.[7] But for the most part, the words "object" and "actor" are synonymous.

An object can be thought of as consisting of two parts: its knowledge and the set of messages to which it can respond. The concept of messages is discussed in the next section. An object's knowledge consists of those objects in the system with which it is acquainted.

The expressive power of Smalltalk comes from making **everything** in the system an object. This makes the language both conceptually simple and elegant. By using a simple but powerful metaphor exclusively, Smalltalk has achieved a level of expressive power far greater than the power of the initial metaphor.

This increase in power can be seen in other languages as well, the most notable of which is LISP. LISP (in its purest form) consists of only two data types that are used to represent both programs and data. The object in Smalltalk is used in a manner similar to the atom and list in LISP: as a universal representation mechanism.

5.2. Message Sending.

Objects are autonomous entities. The user cannot manipulate objects as in other languages. Tasks are done in Smalltalk by asking objects to perform those tasks. This implies that there must be a way to communicate with objects. The paradigm used is message sending.

---

[7] A defense of this statement appears in chapter 6, "The Smalltalk Metaphor"; it is more appropriate to the discussion there.

A message is a request for action, together with any additional information needed to carry out the action. Messages are sent to objects. The object to which the message is sent is referred to as the receiver. The receiver can respond either by changing its own knowledge or by sending other messages to other objects (or some combination of these).

All message sends return a value in Smalltalk. This is the only way that the sender of the message can know that the message has been received and the request honored. It is also the only way to get information from an object.

A message send is specified by giving the name of the object to whom the message is to be sent, followed by the message expression. The message expression is a representation of the message and arguments being sent to the receiver. Smalltalk messages are syntactically divided into three categories: unary, binary, and keyword.

Unary message expressions consist of the name of the message (any arbitrary identifier, like *size* or *print*). Unary messages group left to right, so the expression

anArray size print

sends the message *print* to the result of sending the message *size* to the object *anArray*. The size of the object *anArray* is printed.

Binary message expressions consist of a binary selector (any of a set of special symbols such as "+", "-", and ",") and a single argument. The argument to a binary message can be the name of an object or a message expres-

sion consisting of unary messages. Binary messages also group left to right. The expression

```
3 + 4 * 5
```

sends the message "*" with the argument "5" to the result of sending the message "+" with the argument "4" to the object "3". There is no precedence of operators in Smalltalk.

A keyword message expression is written as a sequence of keyword, argument pairs (where a keyword is an identifier followed by a colon, like *at:* or *new:*). Arguments to keyword messages can be any expression not containing keyword messages. For example, the message

```
anArray at: index + 1 put: (anArray at: index)
```

sends the message *at:put:* to the object *anArray*. This message has two arguments. The first is the result of sending the message "+" to the object *index* with the argument "1". The second is the result of sending the message *at:* to the object *anArray* with an argument of *index*. This has the effect of copying the element at the *index*th position in the array *anArray* to the *index* plus first element of the array. Arguments to all three types of message sends can be parenthesized message sends of any type, as in the example above.

Although there are three syntactic types of message sends, semantically, all three are treated the same way. When a message is sent to an object, the selector (the name of a unary message, binary selector, or the concatenated keywords) is used to look up a method of handling the message. This method is then executed to effect the desired result. If no method can be found, the

user is informed of the problem and given a chance to fix the problem on the fly, or just abort the whole operation.

5.3. Methods.

A method is the Smalltalk representation of the response to a message. Methods are created by the Smalltalk compiler when it is given Smalltalk code. A method contains instructions to change the receiver's knowledge and to send messages to other objects (resulting in the execution of other methods).

Methods are also objects, just like everything else. This makes it extremely easy to handle execution errors. Smalltalk has a symbolic debugger that is entered whenever an error occurs. The debugger is nothing more than an interface to the execution stack formed by the nested method invocations. Since methods understand several messages requesting information about their current state (ie. the state of their execution), the user can make changes to this state of execution. Messages can be sent to open code editors, recompile, back up and single step through the execution, proceed with the execution with the changes made, or simply abort the execution of the methods.

The fact that methods are objects has another advantage. It is easy to build control structures from smaller blocks of code. For example, selection between two blocks of code can be made by sending the message *ifTrue:ifFalse:* (or a related message such as *ifTrue:*, *ifFalse:* or *ifFalse:ifTrue:*) to a boolean. The arguments to these messages are all blocks

of code, and the method in the class of the boolean will ask the appropriate block to execute itself.

The code passed to control structures is usually very specialized. Since it is not useful to create a separate method (with an associated message) for each such block of code, Smalltalk allows the creation of unnamed methods. These are called "blocks". A block can take arguments, as can normal methods. Syntactically, they are represented as a list of arguments and a list of message sends, separated by a vertical bar and surrounded by square braces. (Examples of blocks appear in appendix B.)

It is also possible to implement iteration in Smalltalk using blocks. The message *whileTrue:*, which is sent to a block, executes its argument as long as the receiver evaluates to true. The message *do:* corresponds to a for loop in procedural languages. It is sent to collections of objects to evaluate its argument, with each element of the collection being used as an argument to the block.

5.4. The Concept of Class.

The class is Smalltalk's answer to the second goal (changeability). Every object is an instance of a class. The behavior of an object is determined by its class. Classes provide a mechanism for data abstraction by describing the behavior of a group of objects.

Conceptually, each object is represented by that object's knowledge of the world and the set of messages to which the object can respond. For example, a complex number is represented by the knowledge it has of its real

and imaginary parts, and the operations that can be performed on it. We do not want to have to describe this behavior for each complex number in the system since it is always the same. Instead, we want to define it once, and have this information available to all complex numbers.

This is exactly what classes allow. A class is a description of the behavior and knowledge common to a set of objects (known as instances of the class). When a message is sent to an object, its behavior is determined by the interpreter which determines the class of the object, and looks in the class description for the method associated with the selector of the message.

Classes can also provide knowledge that is known by all its instances. An example of this would be the value of pi, which is known by all real numbers.

Classes are themselves just objects in the system, and can be sent messages. Typical class messages create new instances of the class and provide public access to class knowledge.

Since all objects are instances of a class, classes must also be instances of a class. The class of a class is called a metaclass. When an object is sent a message, the method is looked for in the class of the object. Since we want all classes to behave differently (consider instance creation), there must be a different metaclass for each class in the system.

Metaclasses are also objects, and so must be instances of a class. Fortunately, there is nothing we expect a metaclass to do other than provide a description of the behavior of its single instance. Therefore, we can create a single class (called *Metaclass*) that will be the class of all metaclasses. (The

class *Metaclass* also has a metaclass, which is also an instance of the class *Metaclass*.) A picture of the relations between classes and metaclasses is given in figure 5.1 (adapted from [Gold83]). (Dotted lines indicate superclass relationships, dashed lines indicate instance relationships.)

5.5. Inheritance.

Classes provide an abstraction mechanism, allowing the definition of a behavior to be shared by many objects. But all instances of the class must have exactly the same behavior. It is often common for two classes of objects to share a subset of their respective behaviors. It would be convenient to

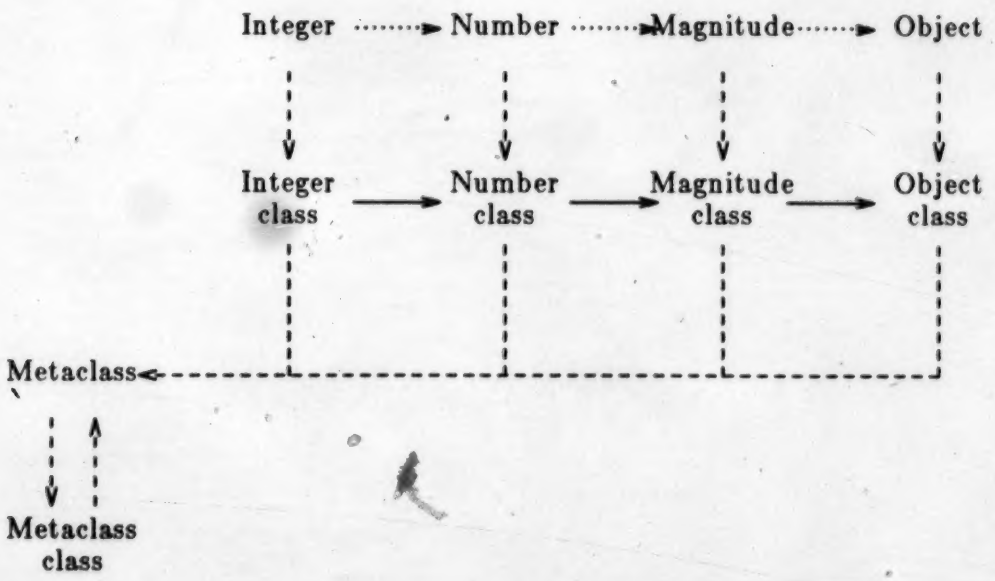

Figure 5.1 -- The Class Structure of Smalltalk.

abstract the commonalities. For example, integers and floating points are both numbers, and share some common operations (such as addition and absolute value). Inheritance provides a means of making this abstraction.

Every class in the system has a single "superclass" from which it inherits part of its own definition. The one exception to this rule is the class *Object*, which has no superclass. The superclass relationship forms a tree-like hierarchy of classes, with the class *Object* at the root of the tree.

Classes inherit from their superclass all messages (and the corresponding methods) together with all the class knowledge of the superclass. Thus, if a message is sent to an object, and the interpreter is unable to find a method for the message in the object's class, the superclass of the class is searched. This process is continued until the root of the superclass tree is found. If no method corresponding to the message is found, the interpreter will respond by sending the message *doesNotUnderstand:* to the receiver of the original message with the original message as the argument. This invokes the run-time debugging system.

In the example of numbers given above, the classes *Integer* and *Float* both share the superclass *Number*. If the message "+" is sent to an integer, then the class *Integer* is searched for a method for "+". If no such method is found, it will search the class *Number*, where it will presumably find the method defined.

The metaclasses also have superclasses. This is necessary if messages sent to classes are to have the same inheritance properties as messages sent to their instances. Furthermore, the metaclasses follow the same superclass

structure as their instances. For example, *Integer*'s metaclass has as its superclass the metaclass of *Number* (since *Number* is the superclass of *Integer*).

The one exception is the superclass of the metaclass of *Object*. Recall that the class *Object* has no superclass, being the root of the class hierarchy. Since there can only be one root in any tree, its metaclass must have a superclass. The superclass of the metaclass of *Object* is the class *Class*, which implements the behavior common to all classes. A more complete picture of the structure of the metaclasses is shown in figure 5.2 (adapted from [Gold83]).

In this way, sending a message to a class will cause the system to look for a method for the message in the metaclass of the class. If no method is found, the metaclass of the class's superclass is searched. If none of the metaclasses has an appropriate method, the class *Class* will be examined, and eventually, through *Class*'s superclasses to the class *Object*.

Subclasses can be used to define refinements of existing classes. For example, a symbol (in Smalltalk) is defined to be a unique string of characters. In other words, a symbol is just a string that requires extra processing. This is represented in Smalltalk by making the class *Symbol* a "subclass" of the class *String*.

5.5.1. Collection Classes: An Example of Inheritance.

The superclass hierarchy of classes often involves several levels. An example of where this is useful is the structure of those classes in Smalltalk

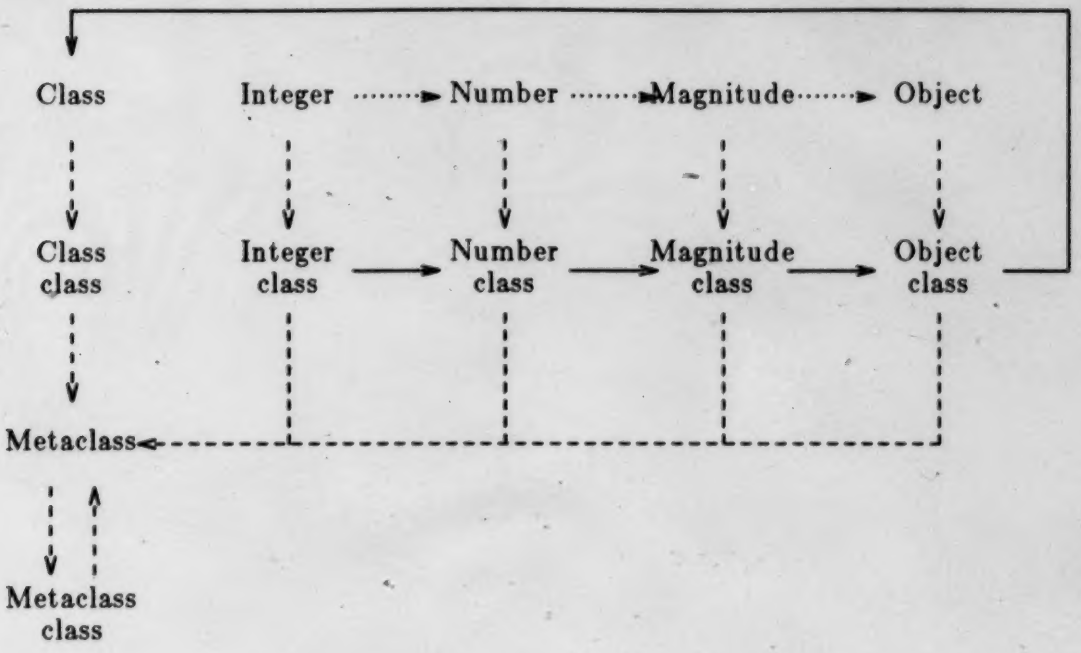

Figure 5.2 -- Full Metaclass Hierarchy.

that represent collections of other objects. All collections of objects share a common minimal behavior. This is represented in Smalltalk's hierarchy by the single class *Collection*, whose superclass is the class *Object*.

The class *Collection* has four subclasses. The class *SequenceableCollection* defines the behavior of collections whose elements can be accessed sequentially. Both *Collection* and *SequenceableCollection* are examples of "abstract" classes, that is, classes defined only to abstract out the commonalities of their subclasses. They do not provide enough behavior by themselves to make instances of themselves useful, though it is possible to create

instances of them. The class *Bag*, another subclass of *Collection*, defines the behavior of an unordered collection that may contain duplicate elements. *Bag* is a "concrete" class, because its instances are fully functional. The other two subclasses of *Collection* are *Set*, which defines the behavior of unordered collections that do not contain duplicate items, and *MappedCollection* which allows indirect referencing of keyed collections by defining a mapping from an external set of keys to the keys used in the original collection.

The class *Set* has a single subclass called *Dictionary*, which is defined to be a set of associations between keys (usually symbols) and their values. The class *Dictionary* also has a subclass, called *IdentityDictionary*, in which keys used to access the values must be identical to the key stored, as opposed to merely having the same structure.

The class *SequenceableCollection* is the only other subclass of *Collection* to have subclasses of its own. It has four subclasses: *LinkedList*, *ArrayedCollection*, *Interval* and *OrderedCollection*. The class *LinkedList* defines the behavior appropriate to lists of elements. Its single subclass, *Semaphore*, defines objects that represent lists of processes. The class *Interval* defines the behavior of collections of numbers representing a mathematical progression. The class *OrderedCollection* defines the behavior of collections whose elements are ordered sequentially. *SortedCollection*, the only subclass of *OrderedCollection* allows the user to define the ordering procedurally. This ordering is then maintained automatically, so that the elements of the collection are always sorted.

The largest subclass of *SequenceableCollection* is the class *ArrayedCollection*. This class implements the behavior of collections whose elements are indexed by integer values. It has six subclasses: *Array*, *Bitmap*, *RunArray*, *String*, *Text* and *ByteArray*. As these data types are generally well known, we will not provide a detailed discussion of them here. The hierarchy of the collection classes is given in figure 5.3.

The above paragraphs have described briefly the structure of the class hierarchy representing the collection classes of Smalltalk. In order to make

---

```
Object
    Collection
        SequenceableCollection
            LinkedList
                Semaphore
            ArrayedCollection
                Array
                Bitmap
                    DisplayBitmap
                RunArray
                String
                    Symbol
                Text
                ByteArray
            Interval
            OrderedCollection
                SortedCollection
        Bag
        MappedCollection
        Set
            Dictionary
                IdentityDictionary
```

Figure 5.3 -- The Smalltalk Hierarchy of Collection Classes.

---

this discussion complete, we now give an example of a message inherited by all collection classes: the message *includes:*. *Includes:* is implemented in the class *Collection*. When sent to a collection, it will answer *true* if the collection contains its single argument, and *false* otherwise. It makes use of the message *do:* which must be defined for all concrete collection classes. The message *do:* allows iteration over all of the elements of a collection by evaluating its block argument for each element of the collection. *Includes:* tests each element in turn, remembering if it finds an element equal to its argument.

If the message *do:* is not implemented by some subclass, the default implementation in *Collection* will be found. This message causes a run-time error to occur, informing the user that a message that should have been implemented was left undefined. Users defining new collection classes are thus given more information when debugging the definitions. The convention of defining methods for messages in order to provide better run-time error checking and notification is common in Smalltalk.

5.5.2. Problems with Inheritance.

One problem with this hierarchical abstraction scheme is that it does not permit a class to inherit from two other classes unless one of the classes is a subclass of the other. This is a serious deficiency. For this reason, most Smalltalk systems include a set of classes that implement what is known as multiple inheritance. Multiple inheritance permits a class to have more than one superclass, though one superclass is usually chosen to be the "primary" superclass.

Chapter 6.

The Smalltalk Metaphor.

The metaphor used in Smalltalk is, for the most part, that described by Hewitt in his work on actors. The programming methodology used in Smalltalk is, therefore, the same as that used when dealing with actors. While Smalltalk is not the only programming language to support this style, it is probably the best known.[8]

This chapter will look at the metaphor behind Smalltalk. It is necessary to learn to think of objects as actors to fully utilize the power of Smalltalk. Knowing the language syntax is just not enough.

6.1. Objects As Actors.

Objects are essentially actors. They are independent, intelligent entities that are able to do complex tasks. To accomplish these tasks, it is necessary for objects to be cooperative. When a request is made (ie. when a message is sent), the receiver of the request will comply to the best of its ability.

In the last chapter, we qualified our statement that objects are actors by stating that they are less restricted than actors. We now examine that claim, in the context of the Smalltalk metaphor.

In one of his discussions of actors, Hewitt stated that "There is no such

[8] Another example is the language PLASMA, developed by Hewitt [Hewi77].

thing as 'action at a distance'".[9] This precludes the occurrence of side effects. But in Smalltalk there is a message that has a side effect. The message is *become:*, and it causes the receiver and the argument to "change places" in the system, with all references to the receiver being changed to refer to the argument, and vice versa. This means that all objects in the system that used to know about the receiver are modified to know only the argument, and objects who knew the argument will only know the receiver. Furthermore, these changes occur in objects without asking the objects themselves. This is a clear violation of the actor metaphor. Therefore, objects are less restricted than actors.

6.2. Object-oriented Programming.

Thinking of objects as actors responding to messages has an effect on the way in which one solves problems. There are three methodologies commonly used when designing solutions to problems: control-based, data-based, and object-oriented. It will be useful to compare the object-oriented approach with both the control-based and the data-based disciplines in order to gain a better understanding of what we mean by object-oriented.

In a control-based approach, the programmer analyzes the problem from the standpoint of the operations that need to be performed in order to accomplish the task. The control structures are designed first. This includes both statement level control and procedural level control structures. This is followed by the definition of the data types manipulated.

---

[9] [Hewi77] p. 325.

The data-based approach uses the opposite order. The problem is analyzed in terms of the data structures needed. The solution is then stated as transformations performed on that data. The actual control structures used to perform those transformations are not designed until after the data structures have been fully defined.

The object-oriented methodology offers a third approach to problem solving. In this methodology, the programmer begins by defining the actors (or objects) in terms of their behavior. The programmer ignores, at this stage, the questions of control and data structures. The behavior we define consists of the way in which each actor will appear to other actors, that is, the external actions of the actor. Once this has been done, the programmer determines the information needed by each actor to obtain the behavior specified. The last stage of the object-oriented approach is to define the control structures used by the actor to affect its behavior.

In all three of these methodologies, the programmer can use either a top-down or a bottom-up approach. In a top-down object-oriented approach, the programmer begins by defining the single actor responsible for the entire task. This would lead to a determination of those actors that interact with the first actor directly. Each of these actors is defined in a similar way, until *fundamental* actors (actors that do not need to interact with other actors) are defined.

It is equally possible to use a bottom-up object-oriented methodology. Under this scheme, the programmer would begin by defining the fundamental actors. Actors that utilize these actors would then be defined until a single

actor, capable of performing the task in question, has been defined.

When programming in Smalltalk, it is common to use a mixture of top-down and bottom-up methods. The reason for this is that the Smalltalk environment is already rich in predefined classes, and for most problems it is only necessary to define a few new classes to build a solution. Because the world is already populated with a diverse collection of classes, Smalltalk is easier to use and more powerful than it would otherwise have been. This is analogous to standard libraries in traditional languages.

Chapter 7.

Implementation.

We turn now from our description of Smalltalk to its implementation. We stated earlier that the primary reason for examining the implementation of Smalltalk is to determine the subsequent influence of the implementation on the system. However, there is also another reason. Most of the recent efforts associated with Smalltalk have been attempts to improve the speed of the implementation [Kras83].

We might well wonder why everyone is trying to improve the implementation. The reason is that Smalltalk tends to be slow. This has been a strong deterrent to the popularity of Smalltalk, and it is sometimes felt that Smalltalk will not be widely accepted, despite its usefulness, until there are better implementations. Improving the speed of Smalltalk is, therefore, seen as a vital effort.

Why is Smalltalk slow? There are three major reasons. First, Smalltalk is currently an interpreted language, and interpreters tend to be slow. Second, Smalltalk is graphically oriented, which requires extensive processing. Third, Smalltalk is a non-VonNeuman language running on VonNeuman architectures. While none of these reasons would be enough in and of itself, when the three are taken in combination the result is a system that is perceived as being slow. Even the fastest implementation tends to be barely fast enough.

It does not seem that the reasons given above are enough to totally account for the perception of slowness, though they certainly contribute to it. The Smalltalk system seems slow to people because Smalltalk is so powerful. By allowing a more natural means of expression, it simplifies the encoding of the programmer's concepts. This increases the speed of the problem solving process so that instead of spending a lot of time typing, the user spends a small amount of time interacting and a small amount of time waiting. In other words, Smalltalk increases the efficiency of its users.

There are other problems with the implementation of Smalltalk besides its speed. One of these is the amount of memory required. The native code (both the interpreter and the predefined Smalltalk objects) occupies about 1.5 megabytes, and this does not include anything that the user might add.

The other major problem with Smalltalk's implementation is that it requires special hardware to be run. Specifically, it needs a bit-mapped graphics screen, a three button mouse, and enough peripheral storage to hold the system between executions. These factors tend to make Smalltalk an expensive language to implement.

7.1. A Virtual System.

In an effort to make the Smalltalk system easier to transport between machines, it has been split into two parts: the virtual machine and the virtual image [Kras81]. The virtual machine is quite small and is machine dependent. The virtual image is far larger (around one kilobyte), but is identical on all installations. The relationship of the virtual image to the vir-

tual machine is shown in figure 7.1.

7.1.1. The Virtual Machine.

The virtual machine is a computer architecture designed especially for Smalltalk. It has an instruction set that is tailored toward executing message sends and other Smalltalk peculiar operations. It also has several "system" calls defined which perform I/O operations, primitive arithmetic, etc. The only code an implementer needs to produce is the code to simulate this virtual machine.

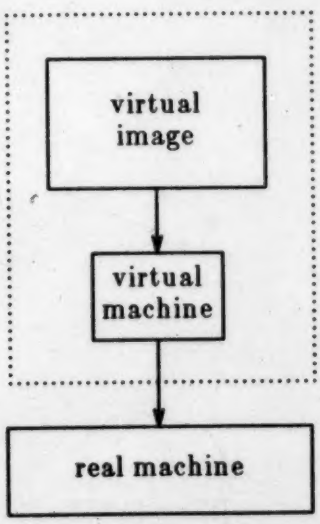

Figure 7.1 -- The Virtual System.

7.1.2. The Virtual Image.

The virtual image is an implementation independent description of all the objects in the system. This includes all the text and pictures, as well as all the Smalltalk code. These objects are stored in a predefined format that is consistent with the original implementation, but that can be easily adjusted to fit alternate implementations.

7.2. Object Memory.

The first major sub-system in the virtual machine is the memory manager [Kaeh81]. The object memory is a heap, from which the space for objects can be allocated, together with the data structures used to manage the heap. The object memory is responsible for allocating and deallocating this space, and for providing access to the space it has allocated.

The first complication to the memory manager is caused by having objects of varying sizes. Because the lifetime of most objects is short, this will tend to fragment memory. This is a well understood problem, and the most common solution is to compact memory at intervals. Unfortunately, with a memory size of well over one megabyte, this tends to be slow. The problem is further compounded when we take into account that most of the object memory contains pointers to objects. These pointers must be modified if the object's location in memory is changed.

The solution taken in most Smalltalk implementations is to use a structure known as an object table. The object table is a mapping between virtual object addresses and logical memory locations. This makes compaction much

faster, since only one entry in the table need be changed, regardless of the number of references to the object being moved.

The disadvantage of using an object table is that it adds a level of indirection to **all** object references. This is another factor which slows down the system.

There are two additional reasons for using an object table. First, it stablizes response time by spreading out the cost of dynamically allocated objects. There is a penalty each time an object's memory is accessed, but this is not noticeable by the average user. If no object table is used, then there is likely to be a noticeable delay whenever the system compacts memory.

There are algorithms that spread the compaction process out over time. They work by doing incremental garbage collection. The use of such an algorithm eliminates the response time argument.

However, there is a more fundamental reason for having an object table. Recall that one message understood by all objects is the message *become:*. This message causes the receiver and the argument to "change places" in the system, with all references to the receiver being changed to refer to the argument, and vice versa. This operation is considerably faster if an object table is used. This message must be sent every time a structure's size is dynamically increased. Depending on the applications of the system the savings could be considerable.

The choice of whether to use an object table is not an easy one to make. It will depend on the types of operations best suited to the underlying

hardware and the type of applications intended for the system. The most common choice is to use an object table. Berkeley Smalltalk is one system that does not make use of an object table. Both the original Xerox implementation and the later Tektronix Magnolia implementation used an object table.

7.2.1. Garbage Collection.

Since the deallocation of objects is transparent to the user, the object memory must provide some form of garbage collection mechanism. We have already seen the effects of compaction on the design of the memory manager, but there is another choice to be made. There are two prevalent techniques for finding unused memory: reference counting and mark-sweep garbage collection.

The preferred method for Smalltalk systems is reference counting. The reason for this is the same as that used to promote the use of an object table. If a reference counting garbage collector is implemented, then the cost of performing garbage collection will be spread out more evenly over the whole execution cycle. This in turn makes the presence of garbage collection more transparent to the user.

The one drawback to reference counting is the possibility of circular data structures, which will never be detected as garbage.[10] Mark-sweep garbage collectors solve this problem, but take a long time, during which the user is unable (typically) to do any work.

---

[10] for a more detailed description of these problems see [Knut73] p.412-413 or [MacL83] p.439-446.

One possible solution is to include both types of garbage collection. When a reference count drops below zero, the space is reclaimed. If the amount of free space drops below some limit, then a mark-sweep garbage collector is invoked to deal with the accumulated circular data structures. This has the advantage that no garbage will go undetected forever, but the extra time required to find it will only be spent if the space is really needed. Also, if a mark-sweep collector is implemented, circular garbage can be removed before the object memory is written out for between-session storage, thus decreasing external storage requirements.

7.3. Bytecodes.

The second major sub-system in the virtual machine is the bytecode interpreter. Bytecodes are the machine instructions for the virtual machine. They are similar to p-code or m-code instructions, consisting of push and pop operations, fetches and stores to object fields, conditional and unconditional branches, and message sending instructions.

Although the choice of bytecodes influences the system, the decisions that must be made when implementing the bytecode interpreter tend to have little or no influence on the system beyond the speed of execution. We will look at the effects of including bytecodes in the definition of the system later in this chapter.

7.4. Primitive Methods.

The third major sub-system within the virtual machine is the collection of primitive methods. A primitive method is like a system call and is

implemented as part of the virtual machine. It performs some task not provided for by the bytecodes. The types of tasks handled by the primitive methods include: integer and floating point arithmetic, input and output, object creation, low level manipulation of objects, execution and process control, and basic system control functions (such as exiting the system, gathering performance measurements, etc.).

7.5. Influence Of Implementation On System.

Our primary motivation for examining the implementation of Smalltalk-80 was to determine the effects of the implementation on the system. We therefore conclude this chapter with a look at those features of the implementation that have the greatest effects on the system, and the effect each feature has.

The first implementation feature that effects the Smalltalk system itself (as opposed to just the performance of the system) is the collection of bytecodes. Bytecodes are optimized for the Smalltalk language. This makes them more efficient for executing Smalltalk code, but it makes them less general than might be desired.

For example, one could write an interactive environment for a programming language other than Smalltalk using the tools available in Smalltalk to build the interface. This situation has occurred with p-code, an intermediate language originally intended for Pascal that has since been used for several other languages. However, it may be difficult to translate the code for the other programming language into bytecodes.

Another problem with bytecodes is that, although they exist within the system, they are not objects. One goal of Smalltalk was to apply the concept of "objects" more universally than had earlier languages. Although compiled methods (blocks of bytecodes resulting from "compiling" Smalltalk code) **are** objects, the instructions themselves are merely "byte encoded". This is done for the sake of efficiency.

It is interesting, however, to note that while instructions are not objects because they are byte encoded for efficiency, that characters, which are also usually byte encoded, are treated as separate objects within the system. A similar treatment could have been applied to bytecodes, resulting in a more uniform system.

Bytecodes are not the only things within the system that are not true objects. Primitive methods are not objects either. It is possible for a method to include, as part of its implementation, a call to a primitive method at the beginning of its code. Any code that follows this call is executed only if the primitive method returns a "failure" (ie. detects an error in the arguments).

Since objects are not allowed to return conditions like SNOBOL [Gris71] and ICON [Gris83], primitive methods are not objects. The user of the system can not even refer to primitive methods except when writing code. This concept of "successful" execution does not appear anywhere else in the system.

One can envision a system in which primitive methods are instances of the class *PrimitiveMethod*. The user would not be allowed, of course, to manipulate the state of the objects indiscriminately, but this scheme would

allow for adding new primitives to a running system and for compiling Smalltalk code into the machine code of the underlying processor. The disadvantage of allowing this is that it would be difficult to test the validity of new primitives. This would lead to a decrease in system reliability.

There is another problem associated with primitive methods. The invocation of a primitive method is not the result of executing an instruction in the compiled method. Instead, it is stored as a bit encoding, along with other encoded information, in a single integer value. When a compiled method is asked to execute itself, it first looks at the encoded information in this integer. If it indicates that a primitive method exists for this message, the method will execute the primitive method instead of itself. This is not as general as it could be.

There are several other places in the system in which other information is similarly encoded. Certain information in class descriptions is also bit encoded. In practice, these bit encodings are not a serious problem. It is possible for the user of the system to access these values, making them appear to be merely integer encodings. While integer encodings may not seem elegant, they can help make the system more efficient in terms of speed.

In general, the question to ask when evaluating an implementation technique is whether it disrupts the user's view of the system as a collection of actor-like objects. We have seen a clear example of a Smalltalk feature that did violate this view of the system. The message *become:* causes a side-effect, which goes against the actor metaphor in its strictest sense. On the other hand, using integer encodings for certain fields of frequently used and

common objects does not violate this principle. Therefore, integer encodings are consistent with Smalltalk (here we mean some ideal form of Smalltalk) while messages like *become:* are not.

Chapter 8.

The Smalltalk Reality.

We have already seen some of the features of Smalltalk, but we will take a more in-depth look at them in order to identify their strengths and weaknesses.

8.1. Where It Falls Short.

We begin our examination with those features that we feel are undesirable. Our examination will encompass three major areas. We will first examine the problems with the syntax and the semantics of the Smalltalk programming language. We will then look at the Smalltalk class definitions that create problems when using Smalltalk. We will conclude by restating the most important problems with the implementation.

8.1.1. Problems With the Language.

8.1.1.1. Cascaded Messages.

We said earlier that Smalltalk message sends consist syntactically of the name of the receiver followed by a message expression. However, the language defines a feature called "cascaded messages". A cascaded message consists of a single receiver, followed by one or more message expressions separated by semicolons. This is a shorthand way of sending each of the messages to the receiver, one after the other, as if each message expression had

been preceded by the receiver's name.

The concept of cascaded messages seems convenient at first, but it violates a primary principle of programming language design: the principle of *syntactic consistency* [Rich77], [MacL83]. Syntactic consistency means that a single concept should have a single representation. In this case, however, a single concept (a message send) has two forms. We do not feel that the convenience of cascaded messages is enough to justify the syntactic complexity and possible confusion that results.

8.1.1.2. Assignment.

The next language feature we look at is assignment. We intentionally postponed introducing assignment until now because of the problems associated with it. Syntactically, it is represented by a variable name followed by a left-arrow (←) and a message expression. Semantically, this means to bind the variable given on the left to the object which is the result of the message send on the right. Also, as in standard programming languages, a variable name occurring within a message send is replaced by the object to which it is bound.

Although an assignment is written like any other message send, it is not. The assignment "operator" is not a binary selector. There are three important differences between it and binary selectors. First, its "receiver" (ie. the left hand side) is not dereferenced as are receivers of messages. It is not possible to specify this special treatment for real binary selectors.

Second, it can not be redefined or overloaded. Binary selectors can have definitions in many different places. For example, the selector "+" indicates addition if sent to a number, and concatenation if sent to a string. There is nothing in the definition of the classes which implement message selectors or methods to prevent the character "←" from being used as a binary selector. It is only the language definition that prohibits this use.

Third, the value "returned" by the assignment can only be used in another assignment or as the value returned by a method. True message sends return values that can be used anywhere, including using the value as the argument to another message send. The use of the assignment "operator" is inconsistent, and should be changed. A combined solution to both this and the next problem will be presented later.

8.1.1.3. Variables.

The last problem with the Smalltalk programming language that we examine in this paper is the definition of variables. The problem is that variables are not objects. All objects are required to be an instance of a class. But variables do not have a class, so they can not be objects. It is also possible to send messages to objects, but it is impossible to send messages to variables.

There would be nothing wrong with this if variables were merely a syntactic convenience for describing objects to the compiler. The period, used between message expressions in a method description to separate expressions, is exactly this. The period is needed by the compiler, and does not pose a

problem.

But variables are more than just names given to objects for reference within a given piece of Smalltalk code. If we can change the object referenced by them, they must have an existence apart from the objects assigned to them. (We are certainly **not** modifying the object represented by them when we assign new values to them). Therefore, we must conclude that variables exist within Smalltalk.

But, as we stated earlier, variable are not objects. The value of a variable is **always** an object, but the variable itself is not. This is another case in which the actor metaphor is not strictly maintained.

It is because variables are not objects that message sends are always call-by-value. They can not be call-by-reference since no reference to the variable exists. We can not pass a variable to a method, since only objects can be passed as arguments. If we could pass a variable, then we could change the value of the variable, thus producing a call-by-reference discipline.

The definition of variables used in Smalltalk is a throwback to the concept of memory cells used in traditional VonNeuman architectures. The cells are used to store objects, but the concept is the same. This is not to say that the idea of variables is somehow bad, only that the concept should be represented in a consistent way.

8.1.1.4. A Solution.

The solution, to both the problem of variables and the problem of assignments, is to make variables be objects. If variables were objects, the inconsistency would be removed. Furthermore, the assignment "operator" could be a true operator (that is, a binary selector) which is sent to the variable.

Variables could be implemented by either defining a new class or by using an existing class (for example *Symbol*). Using an existing class would require adding a new piece of information to the knowledge of instances of the class (namely, the object bound to the instance). The class *Symbol* would work well for this, because symbols are already unique within the Smalltalk system.

Symbols within Smalltalk would play the same role as atoms in LISP. Symbols could also be defined to have an arbitrary number of properties, making symbols totally analogous to atoms. Notice that this would have the extra benefit of allowing the user to experiment with different binding strategies.

8.1.2. Problems With the Class Descriptions.

The next set of problems we wish to examine come from the definitions of the classes supplied with the Smalltalk system. There is only one instance of a class definition that needs improvement, but there are a couple of more general observations about the use of classes as a structuring technique.

8.1.2.1. The Class *BitBlt*.

Since Smalltalk is graphically oriented, it is not surprising to find that a substantial portion of the pre-defined classes deal with graphics. One of those classes is *DisplayScreen*, whose instances represent physical display devices. The class *DisplayScreen* is a subclass of the more general class *DisplayObject*. A display object is an image that can be displayed.

The way in which one performs graphical operations is by sending the message ***copyBits*** to an instance of the class *BitBlt*.[11] Instances of the class *BitBlt* represent bit manipulation operations. They have knowledge about all the primitive operations, including specification of a combination scheme and clipping. The instances of *BitBlt* also know who the bits are to be taken from, and who is to receive the bits.

The class *BitBlt* was created for two reasons. First, there are fourteen separate parameters governing the copy operation. Forcing the user to respecify each of these values for every copy would be unpleasant at best. Typically, several operations are done with only minor changes to some of these arguments. Therefore, the class *BitBlt* allows a way to specify the constant factors only once. The second reason was to allow the creation of a single implementation of this frequently performed task, that could be optimized to improve the speed of Smalltalk. The message ***copyBits*** is implemented as a primitive method.

---

[11] The name *BitBlt* stems from an early implementation of Smalltalk on a machine with an instruction called BLT, which stood for "block transfer". A *BitBlt* operation is a transfer of bit locations, hence the name "BitBlt", or "bit block transfer".

The problem with *BitBlt* is that we are sending a message to an instance of this class to effect a change in another object. *BitBlt* is acting as an interface to one of its display objects. This is a good solution, except that we already have an interface to objects called messages. A message is an object that remembers the operation being requested, and all the arguments to be passed on to the receiver. *BitBlt* is just a special (ie. single) purpose message, and there is no real need for this. Instead, instances of *BitBlt* should be represented as general messages sent to display objects.

There are two possible reasons for making *BitBlt* a separate class. The first is the efficiency question raised above. This issue is a red herring. If bit operations were performed by sending a message to display objects, then the method associated with that message could invoke the optimized primitive. The second reason is that messages are rarely created. When the compiler finds a message send, it translates the send into the bytecode equivalent of the send. This too, is an insufficient reason since messages work just as well.

8.1.2.2. Scalars and Subranges.

We now move on to make some observation about the use of classes as a structuring technique. For most data structures, classes provide a natural and convenient way of expressing the behavior of the data. But there are two categories of data for which classes seem to be less than ideal. These categories are scalars and subranges.

Scalars and subranges are important data types, and the difficulties with representing them are interesting. There are several examples of scalar data

types in the Smalltalk system, but there are no classes that represent subranges. The lack of such classes is significant, especially in light of the frequency with which facilities to define subranges are being included in most modern procedural languages.

The examples of scalar data available illustrate another problem. Consider the class *SmallInteger* (which implements integers within a finite range). Instances of this class are stored as pseudo-objects (although the representation is well hidden) to make them more efficient. Conceptually, all scalars know one piece of information: their ordinal value within their class. This information is typically stored as an integer value. Instances of the class *SmallInteger* are special in that they are their own ordinal value.

Instances of the class *Character*, on the other hand, must explicitly store their ordinal value. The ordinal value is based on the ASCII encoding scheme. Because the ordering of characters is defined in the virtual image, consistency is guaranteed. This also allows characters to be stored compactly as bytes when they are used to compose a string. The class *Character* provides a message that returns the character whose ordinal value is given as the argument.

Characters and integers are conceptually similar. All characters (or integers) are represented as instances of a single class, and have an ordering defined on them. Because of these similarities, it is natural in Smalltalk to abstract the behavior common to them into a single class. The class *Magnitude* provides the behavior common to all scalar types.

In addition to the classes *Character* and *SmallInteger*, which are fairly common pre-defined scalar types, Smalltalk provides two unusual classes. Instances of the class *Time* represent the time of day, to the nearest second. Instances of the class *Date* represent a date (day, month and year), with the earliest date being the first day of the Julian calendar. Both *Date* and *Time* are subclasses of *Magnitude*.

We said earlier that *Magnitude* implements scalar types. However, not all of the subclasses of *Magnitude* are scalars. For example, the class *Float* is a subclass of *Magnitude*. Floating point numbers are not scalar, since they are conceptually not finite. Another example of a non-scalar numeric subclass is *Fraction*, whose instances are, like floating points, not finite in nature.

In addition, there is a glaring absence from *Magnitude*'s hierarchy. Contrary to most modern procedural languages, Smalltalk does not define booleans to be scalars. Booleans are represented as unique instances of the classes *False* and *True*, both of which are subclasses of the class *Boolean*. The reason for having two subclasses is to enable the objects *true* and *false* to respond differently to control messages. For example, the object *true* responds to *ifTrue:ifFalse:* by executing the first argument, while *false* responds by executing the second argument.

Thus, the representation of scalars, and their proper place in the class hierarchy, is inconsistent at best. Instances of classes like *SmallInteger* and *Character* have a knowledge of their ordinal position within the scalar type. Instances of classes like *Float* and *Fraction*, while inheriting all of the behavior common to scalars, are not scalars. Instances of the classes *Time*

and *Date*, while scalars, do not store an explicit ordinal position. They rely on the ordinal positions of their constituent parts to provide their ordering.

The lack of consistency illustrates the problem. In Smalltalk, a class can only be implemented using arrays and records as constructors. This is adequate for non-scalar data types, but is only marginally acceptable for scalar types. It is simply not general enough for subrange definitions. We feel that a more general structuring mechanism needs to be developed.

8.1.2.3. Private Messages.

Another problem with the Smalltalk class definition scheme is the absence of private messages. Currently, if a message is defined for a class or its instances, that message is available to anyone in the system. In order to promote small, modular methods, Smalltalk should provide a scheme for declaring certain messages to be internal to a given class. But this would add complexity to the method look-up scheme, and the benefits gained might not be worth the cost.

8.1.3. Problems With the Implementation.

We have already looked at most of the problems introduced by the implementation, but we feel that two of these problems are important enough to warrant mentioning them again. These two problems are bytecodes and primitive methods. Specifically, neither of these is an object. It is not surprising that including features not represented by objects would introduce problems. We have said before that the power behind Smalltalk comes from its uniform treatment of everything in the system as an object. Introducing

features that are treated differently can result in a loss of expressive power.

8.2. Where They Did It Right.

We turn now to the more positive aspects of Smalltalk. What follows is, of necessity, general. The problem here is that there are so many good features that picking out a reasonable number for discussion is impossible without leaving out something important. Even so, there are a couple of general classes of benefits that we will discuss to make this discussion more balanced.

8.2.1. Benefits of the Class Descriptions.

One benefit of using Smalltalk is that the system provides a rich set of building blocks. As we said before, implementing a solution to most problems of moderate size involves defining a handful of new classes, or adding to existing classes, and coordinating the actions of the instances of these classes. The rewards are often far greater than the effort expended. This is due to the fact that so many classes are pre-defined.

Smalltalk also promotes sharing of class definitions by providing support for transmission of classes in a standard textual way. This is important in any programming language if we are to avoid re-inventing the wheel every time we write a program. Traditional programming languages that support separate compilation permit this same approach. But traditional languages of this sort do not generally come with as many standardized, pre-defined classes as Smalltalk. This standardization of pre-defined classes is due to the fact that there is only one version of Smalltalk.

Another advantage of having such a large base of classes in the system is that it provides the beginning programmer with many useful examples. Because object-oriented programming is unfamiliar to many people it is important to have a large set of examples. This should help those who are unfamiliar with Smalltalk begin programming in the system.

8.2.2. Benefits of the Metaphor.

The other general class of benefits stems from the underlying metaphor used. The actor formalism, developed by Hewitt and utilized by Smalltalk, provides considerable expressive power. We have argued before that this is primarily because it is both simple and consistent.

However, just being powerful does not mean that the language is useful. It is possible to have a language that is powerful, but difficult to use. Fortunately, this is not the case with Smalltalk. Smalltalk provides a useful way of expressing solutions to problems. Because of the nature of Smalltalk, these solutions also tend to be both simple and elegant.

This does not make Smalltalk a replacement for other languages. There are times when other programming languages may be better. For example, if speed and efficiency are major considerations, then Smalltalk is probably not the best choice. As the implementations of Smalltalk are improved, this will no longer be true.

Another result of using the actor formalism is that Smalltalk is easy to learn. The model of intelligent actors is an intuitive one. This makes Smalltalk easy for non-programmers to use. Because Smalltalk is interactive,

non-programmers can quickly learn how to write message sends (as opposed to merely using menus). This is the first step in learning to program in Smalltalk.

Smalltalk has also been taught to children with some success. This is not surprising, since Smalltalk has many similarities to LOGO [Pape80], another language used to instruct children. Both languages are graphical in nature, though in slightly different ways. They are both modular but LOGO is procedural, stressing small simple procedures. Both are small and syntactically simple, with correspondingly simple semantics. This is another strong indication that the actor metaphor is a good one for a personal computer like the Dynabook is envisioned to be.

Chapter 9.

Future Directions.

Smalltalk's history has been one of constant improvement. There were eight years and three versions between Smalltalk's conception and the release of Smalltalk-80. This is a large precedent for continued improvements, although the changes may be slower now that Smalltalk has been released. And while there has yet to be a Dynabook, work will undoubtedly continue in one form or another. It is only fitting that we take a look at some of the possible enhancements and future directions for the language and the system.

9.1. Strongly Typed Smalltalk.

It has been suggested that Smalltalk should be made into a strongly typed language.[12] The benefits of strong typing are well known. Although Smalltalk's variables are currently polymorphic, this feature of the language is used in relatively few places. In most of the instances where it is used, variables are allowed to be of any type that is a subtype of some single class of objects. It is almost never the case that a single variable will have values that do not share a common set of messages.

The type of an object in Smalltalk would be the class of the object. Variables would be allowed to have values that are objects whose class is either the same as or a subclass of the "type" of the variable. In addition to the

[12] Possible methods for adding the concept of types to Smalltalk have been discussed before in

knowledge already known by objects, each object would know the "type" of the knowledge known. This would allow the Smalltalk compiler to do the same kind of checking done by the compilers for strongly typed languages like Pascal, Modula-2, and Ada.

9.1.1. Problems with Strong Typing.

There are a couple of problems with introducing strong typing to Smalltalk. First, strong typing reduces the freedom of expression allowed. This is not necessarily a bad thing, the question we need to ask is whether or not this freedom is needed. Since there are few times when this freedom is taken advantage of, it does not seem as though this would be as serious a problem as it first appears to be.

The second problem is a more difficult one to solve. The question is, how do we handle objects that represent collections of other objects? As a typical case, consider the class *Array*. Currently, arrays can have elements of differing types. But if we want Smalltalk to be strongly typed we would need to include the concept of a "base" type for collections. Again, the ultimate question is whether this would inhibit the natural expression of solutions to problems. This is somewhat more difficult to determine.

9.2. From Programming Environment To Operating System.

Another possible future enhancement is make the Smalltalk environment a multi-user one. This would, ideally, make it possible to use Smalltalk for projects requiring teams of programmers. It would also have the benefit of

removing the "toy" language label which is sometimes attached to Smalltalk.

There are many problems with designing a multi-user Smalltalk. One of the first problems is protection of users from other users' changes to the system. Smalltalk is an open system, allowing the user to change anything. This should still be allowed, but changes made by one user should not affect other users of the system unless the other users accept the changes.

Note however, that a multi-user Smalltalk is contrary to the concept of the Dynabook. Recall that the Dynabook was to be a **personal** computer. This implies a single user, who is in complete control of the system. This vision must be balanced against the appeal of using Smalltalk for large projects.

One possible compromise is to use a Smalltalk-defined network to communicate information between two (or more) Smalltalk environments. There is already an implementation independent textual format for transferring class definitions between Smalltalks, but an external mechanism is required for shipping the text from machine to machine.

One solution to this problem is to add a built-in modem or other communication facility to the Dynabook, together with a protocol for transmitting individual objects between machines. We would not need to restrict ourselves to the static loading of objects. One can envision an application which would allow one to communicate with other Dynabook owners by entering information into a "communication" window and having the information be displayed on the other Dynabook's screen (also in a window, of course). This

[Suzu81] and [Born82].

information could consist of words, pictures, music, etc.

9.3. Parallel Computation and Multiple Processors.

A major problem with Smalltalk is its speed. A natural solution to this problem is to use multiple processors. Smalltalk currently supports multiple processes within the system, by time sharing the single processor between them. There are two ways in which multiple processors could be utilized, and some combination will probably used in the near future.

The first way to use more than one processor is to incorporate special dedicated processors to manage the hardware with a single general processor for execution of Smalltalk code. An example of this would be to have a dedicated display processor to update the screen. This would help make up for the slowness of the graphics. It is not at all clear how control or information should be split between these processors, however.

The second way to use multiple processors is to have more than one general purpose processor in the system. These processors could be modeled in the system as instances of the class *Processor*, and would probably communicate by sending messages, with the rest of the objects being "shared". This use fits well with the notion of having independent actors, since it would allow multiple actors to be active at once. As the number of general purpose processors increases, the attractiveness of special purpose processors becomes greater.

9.4. Introducing Color.

Another possible addition to the Smalltalk environment is color graphics. There is a lot of work remaining to be done with respect to investigating and formalizing the use of color in user interfaces. Since Smalltalk makes it easy to create and test alternate styles of user interfaces, it seems an ideal place to make these experiments. Whether color should be used every where, or just in selected areas is one of the questions to be answered.

In addition, recall that the goal of the Dynabook is to provide a medium for personal expression. Color seems to be a required component for such a system. Since Smalltalk is the language of the Dynabook, adding color to Smalltalk seems to be necessary.

9.5. Introducing Sound.

Another component that needs to be added to Smalltalk before it is used in the Dynabook is sound. Early versions of Smalltalk included sound, but it was dropped in more recent versions. This was probably done to make Smalltalk more portable. As more and more machines are produced which incorporate sound generators, sound will once again become an integral part of the system. Applications using sound are already being written for machines that can generate sound, such as the Macintosh.

Chapter 10.

Conclusions.

We have examined the Smalltalk system, and made several observations about it. There are several problems with the system as it is currently defined and implemented. However, Smalltalk is clearly an important system for several reasons. It introduces a new metaphor for problem solving. It provides an interactive graphical environment that improves programmer productivity. It provides a large and extensible base of data types, organized in a hierarchical structure. Its inheritance mechanism allows abstraction of common behavior, localizing modifications to the definitions. And finally, we examined a few of the possible enhancements which could serve to make the Smalltalk system even better.

Smalltalk is the combination of Alan Kay's vision of the Dynabook and Carl Hewitt's actor formalism. The idea behind the Dynabook is an appealing one. The actor model, by its very simplicity and consistency, provides a powerful way of expressing solutions. Together, they have influenced the field of computing science dramatically. It seems likely that they will continue to do so for many years to come.

References.

[Alth81] Althoff, James C., Jr., "Building Data Structures in the Smalltalk-80 System," *BYTE*, pp. 230-278 (August 1981).

[Born82] Borning, Alan H. and Daniel H. H. Ingalls, "A Type Declaration and Inference System for Smalltalk," *Proceedings of the Ninth Annual ACM Principles of Programming Languages Symposium*, (January 1982).

[Deut81] Deutsch, Peter L., "Building Control Structures in the Smalltalk-80 System," *BYTE*, pp. 322-346 (August 1981).

[Fair85] Fairly, Richard E., *Software Engineering Concepts*, McGraw-Hill Book Company (1985).

[Gris83] Griswold, Ralph A. and Madge T. Griswold, *The Icon Programming Language*, Prentice-Hall (1983).

[Gris71] Griswold, R. E., J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall (1971).

[Gold82] Goldberg, Adele, "The Influence of an Object-Oriented Language on the Programming Environment," *(Journal paper)*, pp. 35-54 (1982?).

[Gold84] Goldberg, Adele, *Smalltalk-80 : The Interactive Programming Environment*, Addison Wesley, Reading, MA (1984).

[Gold83] Goldberg, Adele and David Robson, *Smalltalk-80 : The Language and its Implementation*, Addison Wesley, Reading, MA (1983).

[Hewi73] Hewitt, Carl, Peter Bishop, and Richard Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," *Third Annual IJCAI Proceedings*, pp. 235-245 (1973).

[Hewi77] Hewitt, Carl, "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence* **8** pp. 323-364 (1977).

[Inga81a] Ingalls, Daniel H. H., "Design Principles Behind Smalltalk," *BYTE*, pp. 286-298 (August 1981).

[Inga81b] Ingalls, Daniel H. H., "The Smalltalk Graphics Kernel," *BYTE*, pp. 168-194 (August 1981).

[Inga78] Ingalls, Daniel H. H., "The Smalltalk-76 Programming System -- Design and Implementation," *Conference Records of the Fifth Annual ACM Symposium on the Principles of Programming Languages*, pp. 9-16 (January 1978).

[Kade78] Kaden, Neil Ezra, "Understanding Smalltalk," Master's Thesis, University of Toronto (October 1978).

[Kaeh81] Kaehler, Ted, "Virtual Memory for an Object-Oriented Language," *BYTE*, pp. 378-387 (August 1981).

[Kay77] Kay, Alan and Adele Goldberg, "Personal Dynamic Media," *Computer*, pp. 31-41 (March 1977).

[Knut73] Knuth, Donald E., *The Art of Computer Programming, Volume 1 / Fundamental Algorithms*, Addison Wealey, Reading, MA (1973).

[Kras83] Krasner, Glenn, *Smalltalk-80 : Bits of History, Words of Advice*, Addison Wesley, Reading, MA (1983).

[Kras81] Krasner, Glenn, "The Smalltalk-80 Virtual Machine," *BYTE*, pp. 300-320 (August 1981).

[MacL83] MacLennan, Bruce J., *Principles of Programming Languages : Design, Evaluation, and Implementation*, Holt, Rinehart and Winston, New York (1983).

[Pape80] Papert, Seymour, *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, Inc. (1980).

[Rent82] Rentsch, Tim, "Object Oriented Programming," *ACM SIGPLAN Notices* **17**(9) pp. 51-57 (September 1982).

[Rich77] Richard, Fredrich and Henry F. Ledgard, "A Reminder for Language Designers," *SIGPLAN Notices*, pp. 73-82 (Dec. 1977).

[Shoc79] Shoch, John F., "An Overview of the Programming Language Smalltalk-72," *SIGPLAN Notices* **14**(9) pp. 64-73 (September 1979).

[Suzu81] Suzuki, Norihisa, "Inferring Types in Smalltalk," *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, pp. 187-199 (January 1981).

Appendix A.

A Partial Hierarchy of Smalltalk Classes.

```
Object
    Magnitude
        Character
        Date
        Time
        Number
            Float
            Fraction
            Integer
                LargeNegativeInteger
                LargePositiveInteger
                SmallInteger
        LookupKey
            Association

    Link
        Process

    Collection
        SequenceableCollection
            LinkedList
                Semaphore
            ArrayedCollection
                Array
                Bitmap
                    DisplayBitmap
                RunArray
                String
                    Symbol
                Text
                ByteArray
            Interval
            OrderedCollection
                SortedCollection
        Bag
        MappedCollection
        Set
            Dictionary
                IdentityDictionary
```

```
Stream
   PositionableStream
      ReadStream
      WriteStream
         ReadWriteStream
            ExternalStream
               FileStream
   Random

File

FileDirectory

FilePage

UndefinedObject

Boolean
   False
   True

ProcessorScheduler

Delay

SharedQueue

Behavior
   ClassDescription
      Class
      MetaClass

Point

Rectangle

BitBlt
   CharacterScanner
   Pen

DisplayObject
   DisplayMedium
      Form
         Cursor
         DisplayScreen
   InfiniteForm
   OpaqueForm
   Path
```

Arc
    Circle
Curve
Line
LinearFit
Spline

Appendix B.

The Definition of the Class Bag.

```
class name                      Bag
superclass                      Collection
instance variable names         'contents '
category                        'Collections-Unordered'

class comment

    "I am an unordered collection of elements. I store these elements in a
    dictionary, tallying up occurrences of equal objects. Because I store
    an occurrence only once, my clients should beware that objects they
    store will not necessarily be retrieved such that == is true. If the
    client cares, a subclass of me should be created."

class methods

instance creation

    new

        ||
        ↑ super new setDictionary

instance methods

accessing

    at: index

        ||
        self errorNotKeyed

    at: index put: anObject

        ||
        self errorNotKeyed

    size

        | tally |
        tally ← 0.
```

```
        contents
            do:
                [:each |
                tally ← tally + each].
        ↑ tally

    sortedCounts
        "Answer with a collection of counts with elements, sorted by
        decreasing count."

        | counts |
        counts ← SortedCollection
            sortBlock:
                [:x :y |
                x >= y].
        contents
            associationsDo:
                [:assn |
                counts add: (Association key: assn value value: assn key)].
        ↑ counts

    sortedElements
        "Answer with a collection of elements with counts, sorted by
        element."

        | elements |
        elements ← SortedCollection new.
        contents
            associationsDo:
                [:assn |
                elements add: assn].
        ↑ elements

testing

    includes: anObject

        ||
        ↑ contents includesKey: anObject

    occurrencesOf: anObject

        ||
        (self includes: anObject)
            ifTrue:
                [↑ contents at: anObject]
            ifFalse:
                [↑ 0]
```

adding

add: newObject

```
"||
↑self add: newObject withOccurrences: 1
```

add: newObject withOccurrences: anInteger

"Add the element newObject to the elements of the receiver. Do so as though the element were added anInteger number of times. Answer newObject."

```
||
(self includes: newObject)
    ifTrue:
        [contents at: newObject put: anInteger + (contents at:
            newObject)]
    ifFalse:
        [contents at: newObject put: anInteger].
↑newObject
```

removing

remove: oldObject ifAbsent: exceptionBlock

```
|count |
(self includes: oldObject)
    ifTrue:
        [(count ← contents at: oldObject) = 1
            ifTrue:
                [contents removeKey: oldObject]
            ifFalse:
                [contents at: oldObject put: count - 1]]
    ifFalse:
        [↑exceptionBlock value].
↑oldObject
```

enumerating

do: aBlock

```
||
contents
    associationsDo:
        [:assoc |
        assoc value
            timesRepeat:
                [aBlock value: assoc key]]
```

```
private

    setDictionary

        ||
        contents ← Dictionary new
```